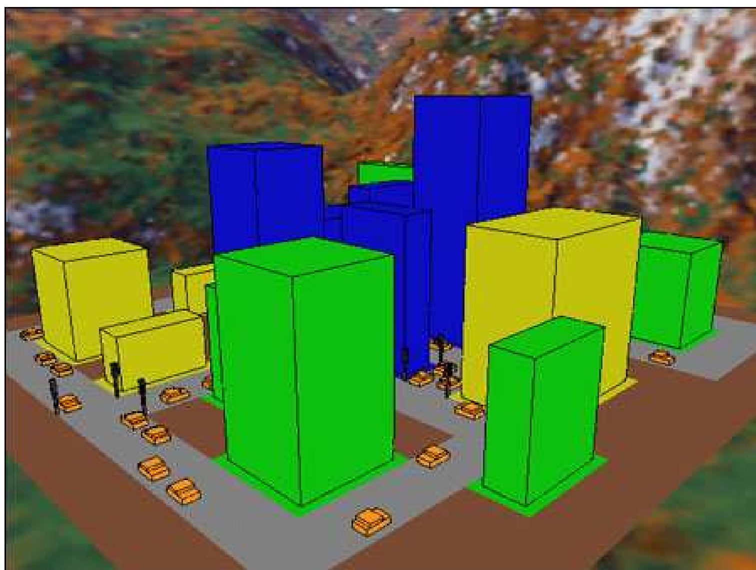
Rapport de Projet : TER

Animation comportementale : Simulateur de trafic routier SimTraffic v1.0 (OpenGL / Glut)

VOURIOT Thierry
Maîtrise d'Informatique
23 mai 2003

ulp
UNIVERSITÉ LOUIS PASTEUR STRASBOURG


SimTraffic v1.0 screenshot

Sommaire

Introduction

Sujet du TER :



Animation comportementale

Le but de ce TER est la réalisation d'un simulateur de trafic routier. Ce simulateur devra être capable de créer et de faire évoluer le trafic au sein d'une ville virtuelle. Celle-ci sera obtenue à partir d'une image bitmap simplifiée la représentant grossièrement. Un premier travail sera donc d'extraire le réseau routier et les différents lieus d'intérêts de la ville de cette image.

Ensuite, il s'agira de simuler l'évolution du trafic en fonction de l'heure qui passe, de manière cohérente avec des règles et des caractéristiques simples définies au niveau de chaque automobiliste (départ, destination, urgence du déplacement…) et de son environnement (feux tricolores, embouteillages,...).

Une visualisation 3D très simple sera nécessaire pour visualiser la ville virtuelle ainsi que les voitures y circulant.

Présentation :

Ce projet reproduit le plus fidèlement possible un trafic urbain idéal : plusieurs véhicules circulent dans une ville virtuelle en respectant certaines règles (le code de la route).
La représentation simpliste des véhicules et des bâtiments permet une excellente vision de l'évolution du trafic.
L'utilisateur peut se déplacer librement dans l'environnement, utiliser des angles de vu fixes ou encore suivre un véhicule en particulier grâce à un système très complet de caméras.
Enfin il est possible d'augmenter ou de diminuer le trafic en temps réel simplement en ajoutant ou supprimant un véhicule.

Quelques libertés ont été prises par rapport au sujet initial puisque le chargement des villes ne se fera pas en analysant une image mais simplement en lisant un fichier. Malheureusement faute de temps les caractéristiques suivantes (départ, destination, urgence du déplacement) n'ont pu être totalement gérés puisque les véhicules se déplacent tous à la même vitesse en prenant une direction aléatoire.

Ce sujet m'a semblé particulièrement intéressant dans le sens où il permet d'appliquer de nombreux thèmes vus théoriquement en cours ; manipulation de vecteurs, de graphes, programmation OpenGL, etc…
De plus la programmation d'un moteur 3D même très simpliste est très enrichissante et constitue un excellent point de départ pour des applications de plus grandes envergures.

Analyse

Outils et environnement de développement :

Le langage de programmation utilisé pour le développement est le C++ car celui-ci permet contrairement au C une programmation orientée objet très adaptée à notre problème. Le Java fut très vite écarté car la programmation 3D avec ce langage n'est pas encore complètement implémentée et reste encore assez lente et gourmande en termes de ressources.

La représentation en trois dimensions utilise OpenGL couplé au toolkit GLUT. GLUT est un ensemble d'outils destinés à faciliter le développement d'applications OpenGL, il permet notamment la gestion des fenêtres, du clavier et de la souris. Bien que d'une puissance limitée GLUT à l'avantage d'être extrêmement facile à programmer, de plus il permet une portabilité à 100% du code (quel que soit le système d'exploitation, tous les programmes seront parfaitement compilables sans aucune modification à apporter).

Logiciels / Compilateurs / Bibliothèques utilisés :

- Microsoft Visual C++ 6.0 Introductory Edition pour la version Windows.
- Gnu g++ pour la version Linux.
- Librairie OpenGL 1.2 (et Mesa 1.2 pour la version Linux).
- Terragen 0.7 pour les textures de paysages.

Choix d'implantation :

Un des points les plus importants du projet fut de bien structurer l'application pour que celle-ci ne devienne pas une « usine à gaz ». Pour cela le découpage en nombreuses classes a été nécessaire.
De plus pour que le code soit le plus compréhensible possible et pour qu'il puisse très facilement être réutilisé dans d'autres projets un nombre conséquent de commentaires est présent.

Pour la représentation des villes, les graphes semblaient être la meilleure solution : on a donc utilisé un graphe non orienté valué. Les intersections représentent les sommets, les arêtes sont les routes de la ville, enfin la valuation représente la longueur d'une route.
Les routes sont toutes à double sens avec une seule voie dans chaque sens.
Pour faciliter le développement on considère que les valuations doivent être entières strictement supérieures à zéro.
Une intersection peut être de deux types différents :

- soit elle est transparente c'est-à-dire qu'il y a seulement deux arêtes qui arrivent dessus (c'est alors simplement un virage).
- Soit il y plus de deux arêtes dans ce cas il s'agit d'une réelle intersection, on ajoute alors un feux tricolore pour chaque route de l'intersection.

Les intersections n'ayant qu'une arête (voies sans issue) ne sont pas prisent en compte.
Le principal avantage de cette solution est de permettre l'application d'algorithmes (comme Dijkstra) sur le graphe qui permettraient de trouver le plus court chemin entre deux positions sur la ville.

La représentation des véhicules utilise une liste simplement chaînée où chaque véhicule représente un nœud de la liste. Cela permet de faciliter les tests de collision puisque chaque véhicule peut simplement connaître la position ainsi que les caractéristiques des autres véhicules en parcourant la liste.

Planning du développement :

Etape	Durée	Description
1	4h	Initialisation OpenGL et GLUT.
2	8h	Création d'une caméra permettant de se déplacer librement.
3	10h	Affichage d'une ville virtuelle (zones et buildings).
4	12h	Chargement d'une ville à partir d'un fichier et représentation sous la forme d'un graphe d'intersections.
5	10h	Création des véhicules (avec un chaînage entre eux) et déplacement dans la ville sans aucun test de collisions.
6	10h	Système de caméras permettant la gestion de différentes caméras : libres, fixes, isométriques et suivant un véhicule.
7	8h	Optimisations et améliorations graphiques du moteur 3D
8	10h	Intersections avec gestion des feux tricolores et adaptation du comportement des véhicules en conséquence.
9	6h	Tests de collisions des véhicules sur une route (entre 2 intersections).
10	6h	Possibilité d'ajouter ou supprimer des véhicules en temps réel.
11	10h	Ajout de la gestion de flots de véhicules sur une route permettant de stopper les véhicules voulant se rendre sur une route saturée.
12	8h	Tests de collisions des véhicules lors du franchissement d'une intersection (priorité à droite).
13	8h	Débuggage.
14	6h	Finalisation du programme (affichage d'informations, aide, amélioration de l'interface utilisateur).

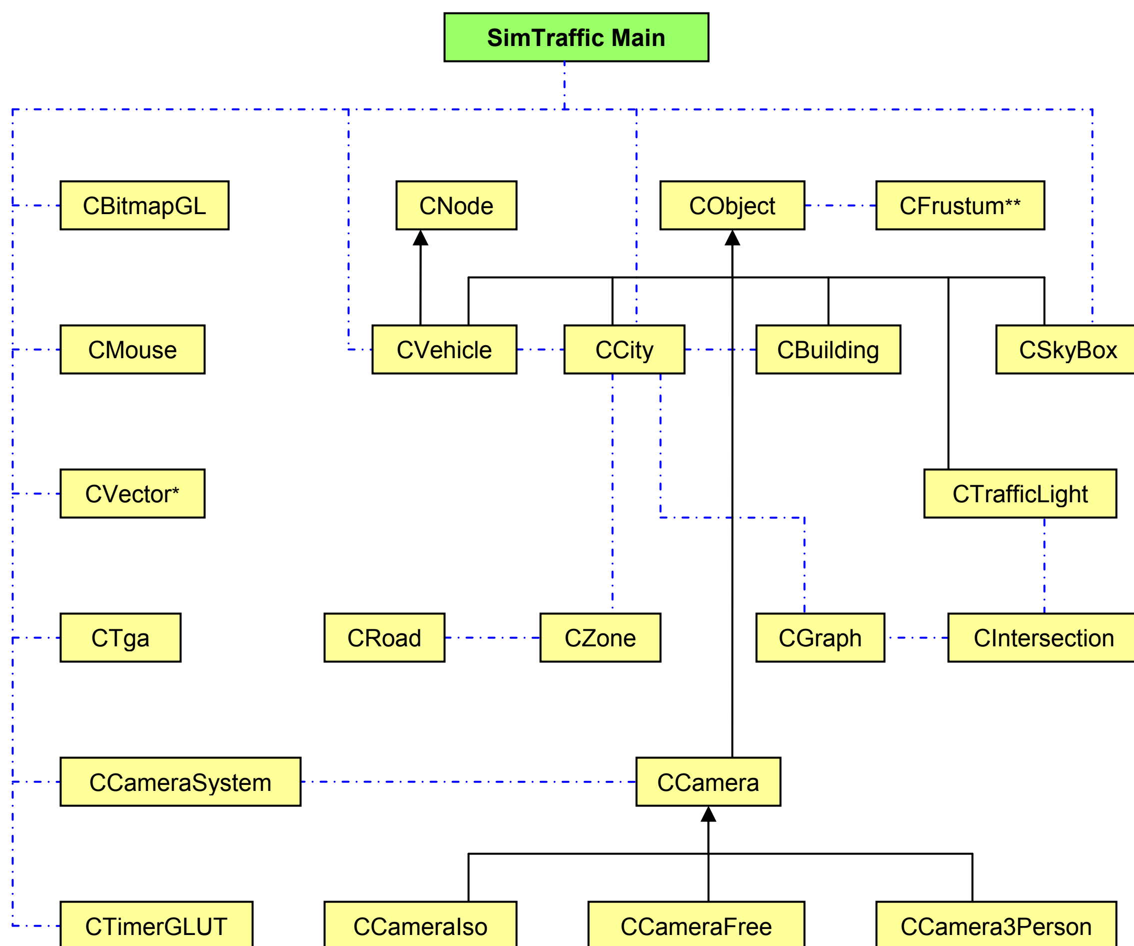
Ce planning présente les grandes étapes de la phase de développement. La durée des étapes n'est qu'une estimation qui prend en compte les différentes recherches et les problèmes techniques rencontrés.

La phase de développement représente un volume horaire d'environ 116 heures, pour la durée du projet il faut ajouter la durée de la phase d'analyse et de la rédaction du rapport.

Développement

Structure globale des classes :

Un ensemble de classes a été écrite pour le développement du logiciel, le schéma suivant montre la structure globale de ces classes.



héritage entre les classes

classes utilisant d'autres classes

* *la classe CVector est quasiment utilisée par toutes les autres classes*

** *la classe CFrustum est utilisée par toutes les classes dessinant des objets 3D*

Descriptions des classes :

`CBitmapGL`

Permet de dessiner des images ou du texte (2D).

`CTga`

Charge ou enregistre une image TGA et crée des textures.

`CTimerGLUT`

Permet de récupérer le temps écoulé et de calculer le nombre d'images par seconde.

`CMouse`

Enregistre les coordonnées et l'état des boutons de la souris.

`CVector`

Représente un vecteur ou un point et définie des opérations vectorielles.

`CCameraSystem`

Permet de gérer un ensemble de caméras.

`CObject`

Un objet contient un identifiant, une position, un nom et un type.

`CNode`

Nœud d'une liste simplement chaînée.

`CVehicle`

Représentation d'un véhicule (affichage et animation).

`CFrustum`

Permet de tester si un objet est placé dans l'angle de vue de la caméra.

`CSkyBox`

Boîte englobant toute la ville et permettant de simuler un paysage.

`CZone`

Une zone de la ville (résidentielle, commerciale, industrielle, route, intersection ou autre).

`CIntersection`

Représente une intersection (contient toutes les routes qui arrivent dessus).

`CGraph`

Graphe des intersections (matrice d'adjacences représentant les routes entre les différentes intersections).

CCity

Représente une ville (fonction de lecture du fichier « ville »).

Ensemble de zones + graphe des intersections.

CBuilding

Représente un building.

CTrafficLight

Représente un feu de signalisation et permet son évolution (vert, orange, rouge).

CRoad

Route entre 2 intersections (à une capacité maximum et flot courant de véhicules dans les deux sens).

CCamera

Caméra fixe.

CCameraFree

Caméra libre pouvant se déplacer et regarder n'importe où.

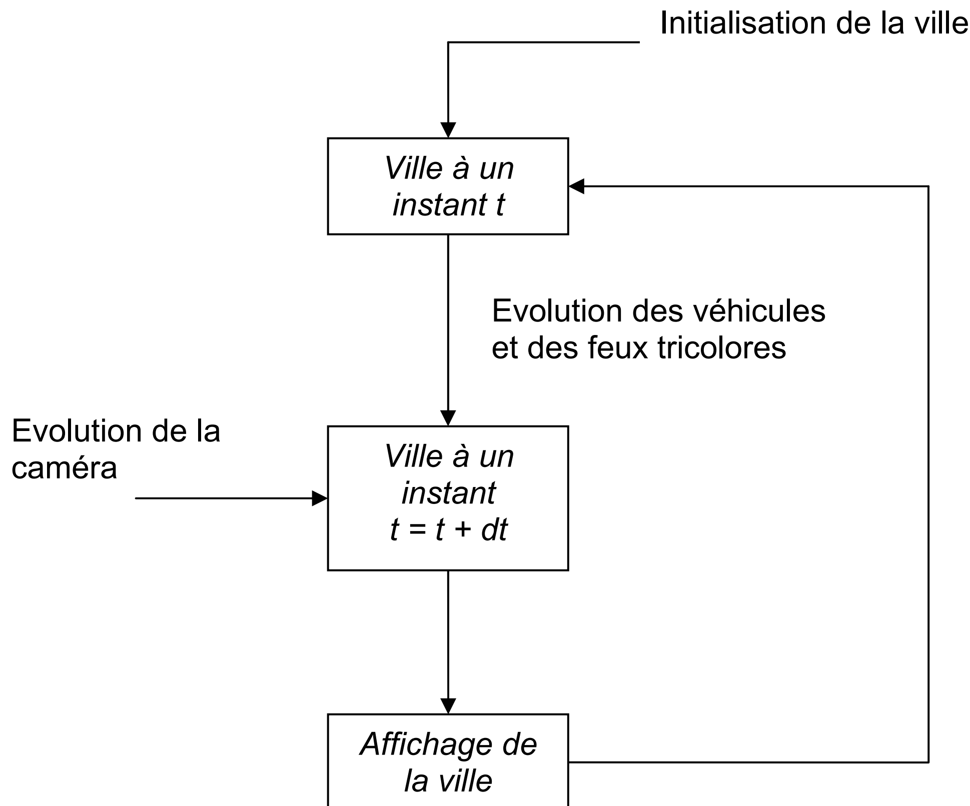
CCamera3Person

Camera placée derrière un véhicule et le suivant.

CCameraIso

Camera placée au dessus de la ville avec un angle légèrement incliné.

Fonctionnement global :



La variable dt est basée sur le temps nécessaire à l'ordinateur pour faire évoluer la ville puis pour l'afficher.
Cette méthode à pour désavantage de faire évoluer les véhicule plus ou moins vite en fonction de la puissance de l'ordinateur.
C'est pour cela que la classe `CTimerGLUT` permet de calculer le nombre d'images affichées par seconde et de limiter ce nombre à 30 (ce qui est largement suffisant pour avoir une bonne impression de fluidité). Par contre sur des ordinateurs moins puissants ce taux peut baisser ce qui aura pour effet de ralentir la vitesse de circulation.

Représentation des villes :

Les villes sont vues comme une succession d'intersections. Tout couple d'intersections forme une route. Une ville est donc un graphe non orienté (R,I), l'ensemble des intersections (I) étant l'ensemble des sommets du graphe et l'ensemble des routes (R), l'ensemble des arêtes du graphe. On appellera graphe de la ville ce graphe.

L'ensemble des intersections est stocké dans un tableau de taille N, avec N le nombre d'intersections présentes dans la ville. L'ensemble des routes est stocké dans une matrice carrée N*N (matrice d'adjacences du graphe de la ville).

Les villes sont divisées en petites zones carrées de taille ZONE_SIZE qui représentent soit :

- une zone libre : FREE
- une portion de route : ROAD
- une intersection : INTERSECTION
- une zone résidentielle : RESIDENTIAL
- une zone commerciale : COMMERCIAL
- une zone industrielle : INDUSTRIAL

La taille d'une ville est donc définie en fonction du nombre de zones en largeur et en longueur.

Les routes sont soit horizontales, soit verticales, c'est-à-dire qu'elles sont dirigées selon l'axe des X ou selon l'axe des Y. Elles correspondent donc à une suite de zones ROAD se suivant sur X ou Y, le nombre de zones donne la longueur de la route.
De plus celles-ci sont rectilignes (pas de gestion des courbures des routes).

Enfin les zones commerciales, résidentielles et industrielles peuvent accueillir des buildings de différentes tailles et hauteurs.

Trois villes ont été élaboré pour les besoins de l'application elles sont visibles dans les annexes.

Le graphe de la ville ainsi que la définition des zones sont stockés dans un fichier `nomville.city`.
La position et la taille des buildings sont stockées dans un fichier `nomville.gcity`.

Format des fichiers `*.city` :

```
[Globals]
largeur hauteur   // taille de la ville (nombre de zones)
nbinters          // nombre d'intersections
nbroads           // nombre de routes
nbzones           // nombre de zones (RESIDENTIAL,COMMERCIAL ou INDUSTRIAL)

[Intersections]
num_inter posX posY A B C D
// numéro de l'intersection, position X et Y, états des intersection
//          C
//          |
//      D--   --B
//          |
//          A
//    -1 = intersection transparente
//    0  = feu rouge
//    1  = feu vert

[Roads]
inter1 inter2     // couple d'intersections formant la route

[Zones]
type posX posY    // type de la zone, position X et Y
// type : 1=RESIDENTIAL, 2=COMMERCIAL, 3=INDUSTRIAL
```

Format des fichiers `*.gcity` :

```
[Globals]
nbbuildings       // nombre de buildings

[Buildings]
type posX posY widthX widthY height
// type du building, position X et Y du coin supérieur gauche du building,
// largeur sur X, largeur sur Y, hauteur
// type : 1=RESIDENTIAL, 2=COMMERCIAL, 3=INDUSTRIAL
```

Le couple (`nomville.city, nomville.gcity`) est chargé au démarrage du jeu.
Il sera tout à fait possible de créer de nouveaux couples pour avoir d'autres environnements de jeu disponibles.

Exemple : Small

	0	1	2	3	4	5	6
0							
1		0		2		4	
2							
3							
4							
5		1		3		5	
6							

- Routes
- Intersections
- Zones résidentielles
- Zones industrielles
- Zones commerciales

Cette ville sera codée de la forme :

```
[Globals]
Small
7      7
6
7
6

[Intersections]
0      1      1      -1     -1     -1     -1
1      1      5      -1     -1     -1     -1
2      3      1      1      0      -1     0
3      3      5      -1     0      1      0
4      5      1      -1     -1     -1     -1
5      5      5      -1     -1     -1     -1

[Roads]
0      1
2      3
4      5
0      2
2      4
1      3
3      5

[Zones]
1      2      2
1      4      2
1      2      3
1      4      3
1      2      4
1      4      4
```

Moteur 3D et caméras :

Les objets 3D héritent tous de la classe `CObject`, cette classe permet de définir pour chaque objet un identifiant, un nom, un type et une position.
Les différents types sont les suivants :

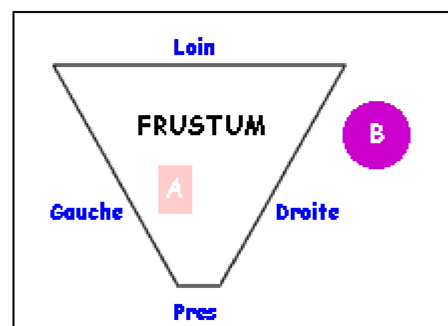
- OBJET : type non défini
- CAMERA : objet `CCamera`
- CAMERA_FREE : objet `CCameraFree`
- CAMERA_ISO : objet `CCameraIso`
- CAMERA_3PERSON : objet `CCamera3Person`
- SKYBOX : objet `CSkyBox`
- CITY : objet `CCity`
- BUILDING : objet `CBuilding`
- VEHICLE : objet `CVehicle`
- TRAFFIC_LIGHT : objet `CTrafficLight`

Les types CAMERA* sont utilisés pour les caméras qui possèdent également les mêmes attributs et qui peuvent être dessinées en mode DEBUG.
Pour la position des objets on utilise la classe `CVector` qui permet de définir soit un vecteur, soit un point.

Chaque objet graphique possède une fonction `Draw(CFrustum *frustum)` qui dessine l'objet en fonction de sa position.
De plus les objets contiennent en général un tableau qui stocke tous les points nécessaires à l'affichage ainsi qu'une fonction permettant la création d'une liste d'affichage (display list : suite de définitions de points qui est désignée par un numéro unique, ce qui permet à OpenGL de la stocker en mémoire et ainsi de l'afficher plus rapidement).

Enfin pour accélérer encore l'affichage on utilise la technique du frustum culling définie dans la classe `CFrustum`. On appelle frustum de vue l'angle sous lequel est vue une scène.

Exemple :
Sur le schéma ci contre, le trapèze représente notre frustum de vue. Loin, Près, Droite et Gauche son 4 des 6 plans du frustum, Haut et Bas n'étant pas représentés.
Selon la technique que nous allons utiliser, on n'affichera que les objets qui sont présents dans le frustum.
Ici, on voit bien que l'objet A est à l'intérieur de ce frustum et la sphère B en dehors. Le frustum culling consiste donc à se passer de l'affichage de la sphère B mais pas de l'objet A.



Les caméras sont simplement implantées à l'aide de trois vecteurs : la position `m_vPosition` (héritant de CObject), la position de l'endroit où elle regarde `m_vView` et un vecteur `m_vUpVector` qui correspond au vecteur perpendiculaire au vecteur de vue et dirigé vers le haut.
Pour déplacer la caméra (fonction `MoveCamera` et `StrafeCamera`) il suffit juste d'additionner le vecteur de déplacement au vecteurs `m_vPosition` et `m_vUpVector`. Pour la rotation c'est un peu plus compliqué, on utilise notamment le cosinus et le sinus de l'angle de rotation pour recalculer le nouveau vecteur `m_vView`.

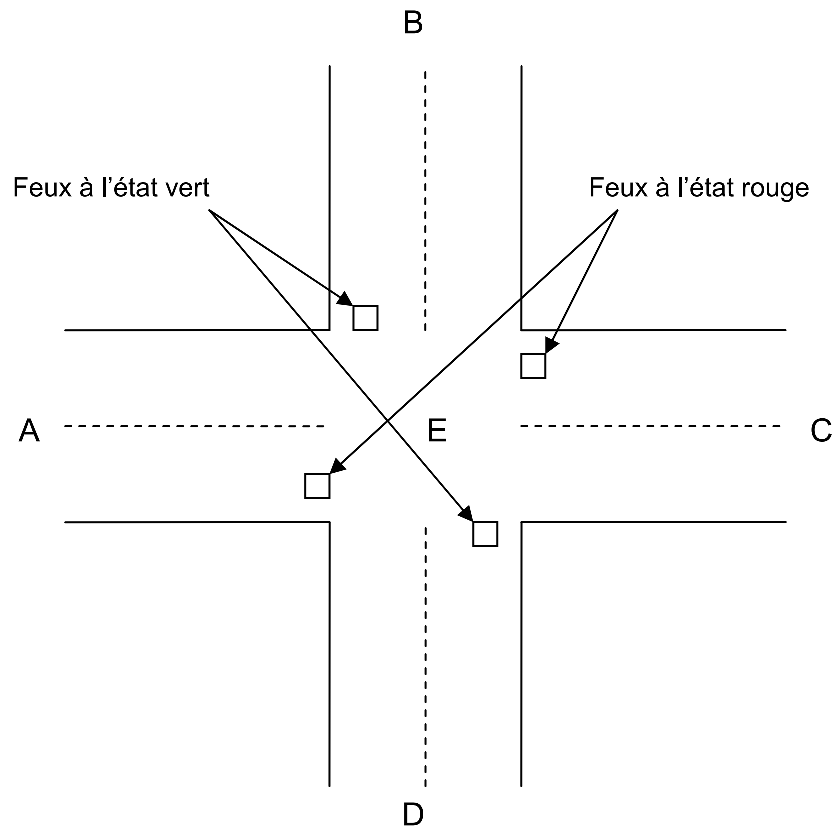
Quatre types de caméras ont été réalisés :

- Caméra Fixed placée à une position et ne pouvant ni se déplacer ni tourner.
- Caméra Free permet de se déplacer librement dans n'importe quelle direction.
- Caméra Third Person permet de suivre un objet (dans notre cas un véhicule) et d'avoir une vue arrière de l'objet.
- Caméra Iso placée au dessus de la ville avec un angle légèrement incliné et ne pouvant pas tourner sur elle-même.

La classe `CCameraSystem` permet de faciliter la gestion d'un grand nombre de caméras. On peut ajouter des caméras au système et en définir une qui sera marquée comme courante. Ensuite grâce à plusieurs fonctions, la navigation entre elles et l'affichage de celle-ci devient très simple à réaliser.

Comportement des véhicules :

Une intersection peut être vue de la façon suivante :



Pour chaque véhicule on connaît (attributs de la classe `CVehicle`) :

- la zone de la ville sur laquelle il est positionné : `m_zoneX`, `m_zoneY`
- sa direction courante X→, Y↓, -X←, -Y↑ : `m_direction`
- sa route courante (couple d'intersections) : `m_inter1`, `m_inter2`
- si il est arrêté : `m_stop`
- sa vitesse : `m_speed`
- le prochain virage qu'il va prendre : `m_nextTurn`
- si il est en train de franchir une intersection : `m_inInter`
- plus d'autres attributs…

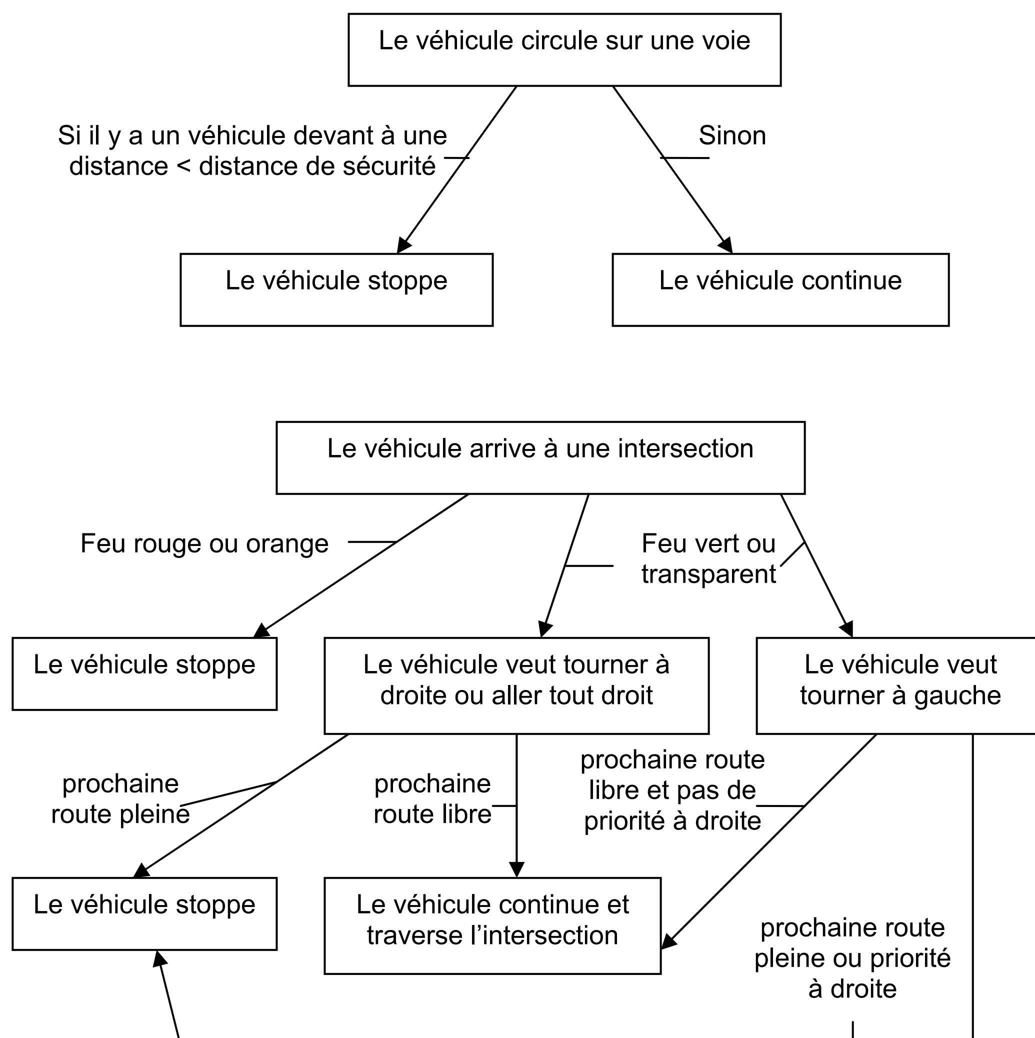
On considère qu'un véhicule ne peut pas s'arrêter dans une intersection et qu'il choisit une direction au hasard lorsqu'il arrive à une intersection.

Chaque route peut accepter un nombre maximal (capacité) de véhicules dans chaque sens. En fait chaque voiture qui prend une route fait augmenter le flot de véhicules dans cette route et lorsqu'elle quitte cette route le flot diminue. Si le flot est supérieur à la capacité, la route ne peut plus accepter de véhicules.

Lorsqu'un véhicule se déplace sur une route, il teste si il y a un véhicule devant lui à une distance inférieure à la distance de sécurité, si tel est le cas le véhicule s'arrête.

Pour le franchissement des intersections, si le feu est rouge ou orange le véhicule s'arrête devant. Si le feu est vert est que le véhicule veut tourner à droite ou aller tout droit, il teste si la route ou il veut se rendre peut l'accepter, si c'est le cas il peut avancer et traverser l'intersection. Si le feu est vert, mais que le véhicule veut tourner à gauche il doit tester si la route l'accepte et si il n'y a pas de véhicules venant en face et allant tout droit ou à droite (priorité à droite).

Comportement :



Interface utilisateur :

L'interface utilisateur est très simpliste car GLUT est plutôt limité dans ce domaine.

Le clavier et la souris sont gérés à l'aide des fonctions `glutKeyboardFunc()`, `glutMouseFunc()` et `glutMotionFunc()` spécifiques à GLUT. La liste des commandes est disponible dans la section manuel d'utilisation.

L'affichage des informations suivantes :

- manuel d'utilisation
- informations sur la ville
- images par secondes
- caméra courante

se fait dans la fenêtre OpenGL à l'aide de la classe `CBitmapGL` spécialement conçue pour l'occasion. Celle-ci crée un contexte 2D avec `glOrtho()` puis utilise `glRasterPos2i(x,y)` pour positionner le curseur et enfin affiche une chaîne de caractères grâce à la fonction `glutBitmapCharacter()`. Cette classe permet aussi l'affichage d'image à l'aide de `glDrawPixels()` et de la classe `CTga` qui permet la lecture du format d'image TGA.

Fichiers du code source :

Chaque classe est composée d'un fichier `.h` contenant la définition de la classe et d'un fichier `.cpp` avec l'implantation des fonctions de la classe.

Commande `wc` sur les fichiers sources (51):

Lignes	Mots	Octets	Fichiers
156	338	3827	bitmapGL.cpp
87	218	2193	bitmapGL.h
234	661	7588	building.cpp
85	247	2302	building.h
138	492	3894	camera3person.cpp
70	222	1921	camera3person.h
116	393	3795	camera.cpp
190	846	6332	camerafree.cpp
61	226	2173	camerafree.h
93	333	3223	camera.h
139	542	4471	cameraiso.cpp
57	216	1988	cameraiso.h
130	306	2890	camerasystem.cpp
86	211	2026	camerasystem.h
385	1276	10383	city.cpp
100	249	2238	city.h
60	110	1105	error.cpp
27	61	639	error.h
228	1371	8473	frustum.cpp
54	200	1731	frustum.h
91	291	1906	global.h
127	303	2683	graph.cpp
75	178	1801	graph.h
124	283	3293	initGLUT.cpp
37	86	939	initGLUT.h
193	480	5230	intersection.cpp
108	352	3330	intersection.h
63	224	1782	makefile
37	76	549	mouse.h
36	60	434	node.cpp
40	64	492	node.h
164	308	3015	object.cpp
114	216	1801	object.h
110	263	2506	road.cpp
81	240	2301	road.h
450	1333	14643	simtraffic.cpp
118	362	3478	simtraffic.h
148	531	4963	skybox.cpp
72	186	1599	skybox.h
376	1696	12508	tga.cpp
127	419	3822	tga.h
106	313	2880	timerGLUT.cpp
71	227	2079	timerGLUT.h
141	323	3356	trafficlight.cpp
64	164	1367	trafficlight.h
125	438	3836	vector.h
695	2290	23027	vehicle.cpp
189	588	5196	vehicle.h
94	236	2321	zone.cpp
71	196	1917	zone.h
6743	**21244**	**192246**	**total**

Principales difficultés :

Dans un projet de cette ampleur il est normal de rencontrer un grand nombre de difficultés, les principales auxquelles j'ai été confronté sont décrites brièvement ci-dessous :

- Ce projet fut ma première application utilisant le langage C++. Bien que j'aie une certaine expérience dans la programmation objet avec Java, il subsiste tout de même de nombreuses différences entre ces deux langages ce qui m'a fait perdre un temps non négligeable lors du démarrage du développement.

- Bien que j'aie déjà développé quelques petites applications en OpenGL, la programmation du moteur 3d et des caméras m'a également posé de nombreuses difficultés. Avec notamment la manipulation des vecteurs, l'affichage de textes et quelques diverses problèmes liés à OpenGL.

- La représentation de la ville fut à son tour une source de nombreux problèmes. Car même si la représentation semblait simple au départ, au fil du développement des classes sont apparues et il a fallu essayer de garder une certaine cohérence entre ces nouveaux objets.

- Enfin le comportement des véhicules a été problématique car de nombreux tests ont été nécessaires et la phase de tests et de débuggage fut plus longue que prévue initialement.

Configuration minimale requise :

- Pentium II 400Mhz
- 64 Mo de mémoire
- Carte vidéo accélératrice OpenGL fortement conseillée (par exemple 3dfx, Nvidia TNT, GeForce, ATI Radeon, etc…)
- Windows 32 bits (9X/NT/2000/XP) ou Linux + Mesa 1.2

Pour lancer l'application il suffit d'exécuter le fichier `simtraffic.exe` (Windows) ou `simtraffic` (Linux).

```
simtraffic(.exe) <nomville>
```

Sans argument le logiciel lance un menu console permettant de choisir entre trois villes de base (small, medium, big).
L'argument `nomville` permet de charger une ville spécifique dont les fichiers `nomville.city` et `nomville.gcity` doivent être placés dans le répertoire `maps`.

Définition des touches utilisables du clavier ainsi que les boutons et les mouvements de la souris : (ce manuel est accessible dans le logiciel en utilisant la touche *h*)

left ou *q* :	déplace la caméra vers la gauche.
right ou *d* :	déplace la caméra vers la droite.
up ou *z* :	déplace la caméra vers l'avant.
down ou *s* :	déplace la caméra vers l'arrière.
bouton gauche de la souris :	effectue une rotation de la caméra.
c :	change la caméra courante (7 caméras différentes : libre, fixe 1, fixe 2, fixe 3, fixe 4, isométrique et suivant un véhicule).
+ ou = :	ajoute un véhicule aléatoirement sur une route de la ville (le nombre maximum de véhicules est limité en fonction de la taille de la ville).
- :	supprime un véhicule (il reste au minimum un véhicule sur la ville).
p :	arrête ou fait redémarrer tous les véhicules.
i :	affiche des informations sur la ville (nom, nombre de buildings, nombre de véhicules, temps écoulés).
t :	affiche ou cache les lignes noires autour des objets graphiques.
f :	affiche ou cache le nombre d'images par secondes.
m :	mode d'affichage plein ou filaire.
echap :	quitter.

Ressources

Bibliographie :

- Programmer en langage C++ 5ème édition, Claude Delannoy, Eyrolles.
- OpenGL 1.2, Mason WOO, Jackie NEIDER, Tom DAVIS, Dave SHREINER, 2002, CampusPress.
- OpenGL Game Programming, Mark J.Kilgard, Kevin Hawkins, Dave Astle, André LaMothe, PrimaTech

Sites Internet :

- NeHe Productions (OpenGL), http://nehe.gamedev.net
- Game Tutorials, http://www.gametutorials.com
- Games Creators Network, http://www.games-creators.org
- CPP France, http://www.cppfrance.com
- Divers tutoriaux OpenGL

Jeux :

- Midtown Madness (Microsoft).
- La série des SimCity.

Conclusion

Le projet fut très enrichissant puisqu'il m'a permis de développer une application complète en suivant quelques parties majeures de la création d'une application :

- Assimilation des techniques de programmation C++ et OpenGL.
- Gestion du projet et partage des tâches.
- Recherche sur le domaine visé.
- Analyse pour une bonne structuration du projet.
- Développement de l'application.
- Phase de tests.
- Rédaction du manuel et du rapport.

Beaucoup d'améliorations sont bien entendu possible, comme par exemple :

- Ajout de véhicules ayant un point de départ et de destination fixés par l'utilisateur.
- Ajout de véhicules ayant différentes vitesses (camion, motos).
- Gestion d'intersections avec des priorités à droite, des céder le passage ou des stops.
- Evolution de la densité du trafic en fonction de l'heure.
- Améliorations graphiques (placage de textures, gestion de l'éclairage, meilleure modélisation des bâtiments et véhicules,…).
- Création d'un éditeur de villes permettant la création des fichiers « villes » de façon graphique avec une interface utilisateur.
- …etc…

Au terme du projet, je peux dire que je suis plutôt satisfait puisque le logiciel fonctionne très bien ; le moteur 3D bien que simpliste suffit largement aux exigences du sujet et le comportement des véhicules semble ne souffrir d'aucun problème majeur.

Par contre je suis un peu frustré par le fait de n'avoir pas implanter de nombreuses fonctionnalités (voir liste ci-dessus) qui me paraisse extrêmement intéressantes, je pense d'ailleurs que je vais continuer à développer cette application durant les prochains mois.

Informations personnelles

Projet (travail d'étude et de recherche TER) réalisé par Thierry VOURIOT du 20 février 2003 au 23 mai 2003 dans le cadre de la maîtrise d'informatique de l'université Louis Pasteur de Strasbourg (67).

Me joindre :

yeri@fr.st

thierry.vouriot@free.fr

Site personnel :

http://www.yeri.fr.st

http://thierry.vouriot.free.fr

Responsable du TER : Olivier Genevaux (genevaux@dpt-info.u-strasbg.fr)

Annexes

Villes disponibles :

Trois villes ont été créées pour les besoins de l'application :

- Routes
- Intersections
- Zones résidentielles
- Zones industrielles
- Zones commerciales

<u>Ville : Medium</u>

Taille : 15 * 14
Nombre de zones totales : 210
Nombre d'intersections : 15
Nombre de routes : 18
Nombre de buildings : 15

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0															
1		**0**			**4**										
2										**9**				**12**	
3															
4		**1**			**5**			**7**		**10**					
5															
6															
7										**11**				**13**	
8															
9			**2**		**6**										
10															
11															
12			**3**					**8**						**14**	
13															

Ville : Small

Taille : 7 * 7
Nombre de zones totales : 49
Nombre d'intersections : 6
Nombre de routes : 7
Nombre de buildings : 2

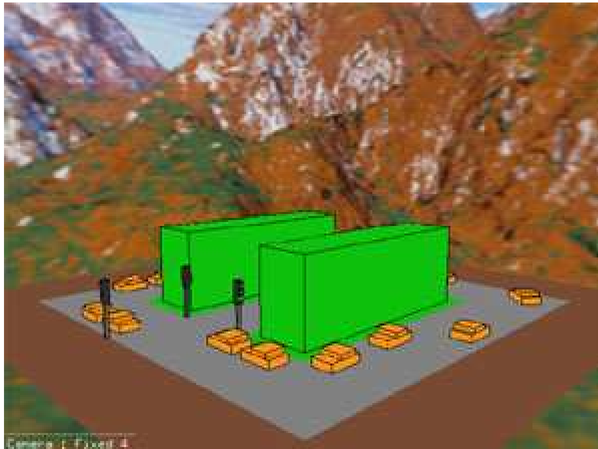
	0	1	2	3	4	5	6
0							
1		**0**		**2**		**4**	
2							
3							
4							
5		**1**		**3**		**5**	
6							

Ville : Big

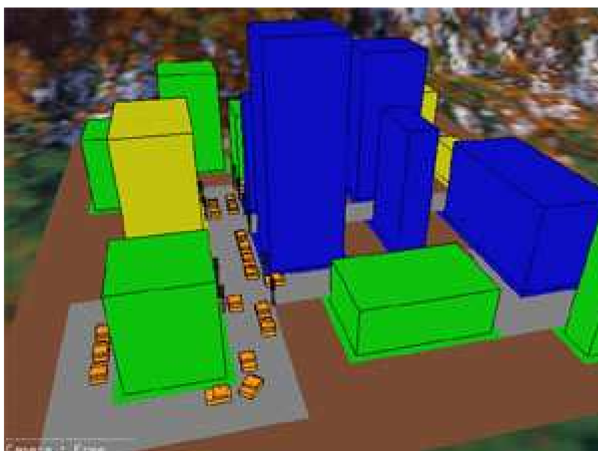
Taille : 30 * 30
Nombre de zones totales : 900
Nombre d'intersections : 49
Nombre de routes : 66
Nombre de buildings : 34

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
0																														
1		0					11						22					31												
2																														
3																		32						39				45		
4																														
5																														
6		1					12						23					33												
7																														
8																														
9																														
10						6					15				26															
11																								40				46		
12		2				7																								
13																														
14						8					16				27															
15																														
16															28				34					41						
17																														
18						9					17				29															
19																														
20			3			10						18							35					42				47		
21																														
22												19			30				36											
23																														
24			4																37					43						
25																														
26			5				13					20		24																
27																														
28							14					21		25					38					44				48		
29																														

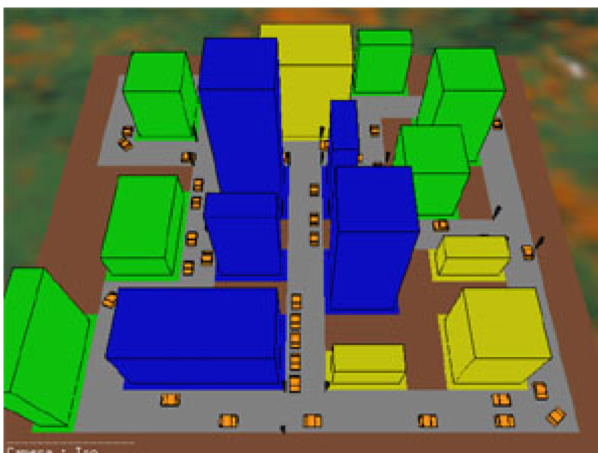
Photos d'écran :



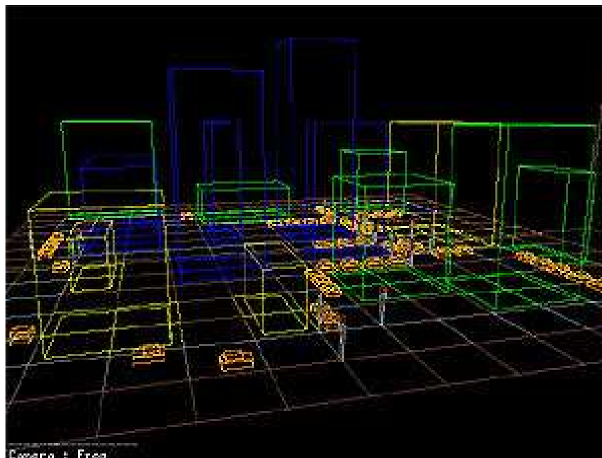
Ville : Small
Caméra : Fixed 4
Mode : Plein
13 véhicules



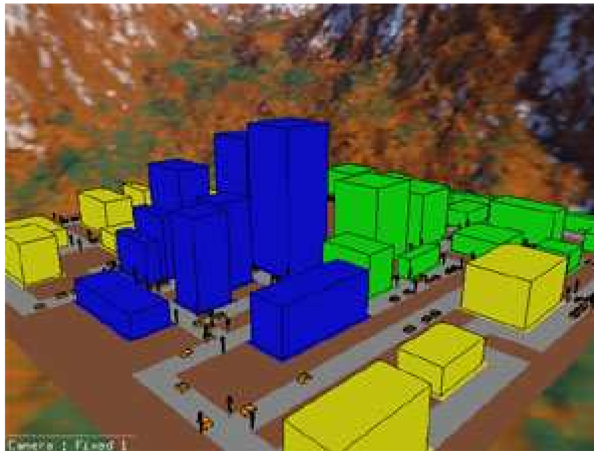
Ville : Medium
Caméra : Free
Mode : Plein
51 véhicules



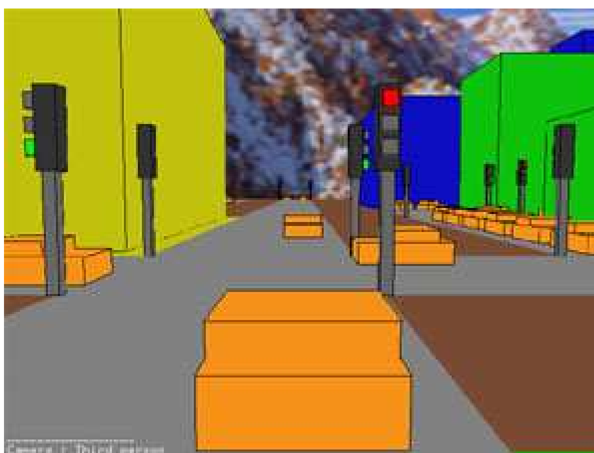
Ville : Medium
Caméra : Isométrique
Mode : Plein
53 véhicules



Ville : Medium
Caméra : Free
Mode : Filaire
53 véhicules



Ville : Big
Caméra : Fixed 1
Mode : Plein
167 véhicules



Ville : Big
Caméra : Third Person
Mode : Plein
203 véhicules